Universidade Federal do Ceará
Departamento de Estatística e Matemática Aplicada
Programação não-linear

Daniel Bruno Juvenal Da Silva
Francisco Claudio Costa Moraes Junior
Herbert Hipolito De Araujo Filho
Renato Marques de Oliveira

Métodos de otimização para problemas não-lineares com restrições

Fortaleza
2020

Daniel Bruno Juvenal Da Silva, 404258
Francisco Claudio Costa Moraes Junior, 403855
Herbert Hipólito De Araujo Filho, 412995
Renato Marques de Oliveira, 414212

Métodos de otimização para problemas não-lineares com restrições

Relatório escrito para a disciplina de Programação não-linear
Prof. Ricardo Coelho

Fortaleza
2020

Sumário

1 Método de penalidade

A teoria e os métodos gerais para problemas restritos empregados ao longo de todo este trabalho seguem a apresentação encontrada em [1]. Os métodos gerais empregados para problemas de busca linear e otimização irrestrita implementados no trabalho anterior da disciplina foram baseados na apresentação encontrada em [2].

1.1 Funções de penalidade

Métodos utilizando funções de penalidade transformam problemas restritos em problemas irrestritos ou sequências de problemas irrestritos ao colocar as restrições na função objetivo a partir de um parâmetro de penalidade que penaliza qualquer violação das restrições. Por exemplo, considere o problema restrito seguinte:

$$\begin{aligned} \text{minimizar} \quad & f(\mathbf{x}) \\ \text{s.a.} \quad & \mathbf{h}(\mathbf{x}) = 0. \end{aligned}$$

Fazendo $\alpha(\mathbf{x}) = \mathbf{h}^2(\mathbf{x})$, este problema pode ser substituído pelo problema irrestrito seguinte, dado que μ seja suficientemente grande:

$$\begin{aligned} \text{minimizar} \quad & f(\mathbf{x}) + \mu\alpha(\mathbf{x}) \\ \text{s.a.} \quad & \mathbf{x} \in \mathbb{R}^n. \end{aligned}$$

Podemos ver que uma solução ótima para o problema acima deve ter $\alpha(\mathbf{x}) = \mathbf{h}^2(\mathbf{x})$ próximo de zero, ou uma penalidade muito grande será incorrida.

Em geral, uma função de penalidade adequada deve incorrer em uma penalidade positiva para pontos inviáveis e nenhuma penalidade para pontos viáveis. Se as restrições são da forma $g_i(\mathbf{x}) \leq 0$ para $i = 1, \ldots, m$ e $h_i(\mathbf{x})$ para $i = 1, \ldots, l$, uma função de penalidade adequada é definida por

$$\alpha(\mathbf{x}) = \sum_{i=1}^{m} \phi[g_i(\mathbf{x})] + \sum_{i=1}^{l} \psi[h_i(\mathbf{x})], \tag{1.1}$$

onde ϕ e ψ são funções contínuas satisfazendo

$$\begin{aligned} \phi(y) = 0 \quad & \text{se } y \leq 0 \quad \text{e} \quad \phi(y) > 0 \quad \text{se } y > 0 \\ \psi(y) = 0 \quad & \text{se } y = 0 \quad \text{e} \quad \psi(y) > 0 \quad \text{se } y \neq 0. \end{aligned}$$

Tipicamente, ϕ e ψ são das formas

$$\begin{aligned} \phi(y) &= [\max\{0, y\}]^p \\ \psi(y) &= |y|^p \end{aligned}$$

1.2 O problema de penalidade

Problema primal

$$\begin{aligned} \text{minimizar} \quad & f(\mathbf{x}) \\ \text{s.a.} \quad & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \\ & \mathbf{h}(\mathbf{x}) = \mathbf{0} \\ & \mathbf{x} \in X, \end{aligned}$$

onde $\mathbf{g}$ é a função vetorial de componentes $g_1, \ldots, g_m$ e $\mathbf{h}$ é a função vetorial de componentes $h_1, \ldots, h_l$, cada uma contínua em $\mathbb{R}^n$ e X não-vazio.

Podemos propor um problema dual onde utilizaremos uma função auxiliar da forma $f(\mathbf{x}) + \mu\alpha(\mathbf{x})$ que nos ajudará a incorporar as restrições diretamente à função objetivo, facilitando a criação de algoritmos.

Problema de penalidade

Dado um conjunto de funções $\theta\colon \mathbb{R}^+ \to \mathbb{R}^n$, $\theta(\mu) = \inf\{f(\mathbf{x}) + \mu\alpha(\mathbf{x})\colon \ \mathbf{x} \in X\}$, onde $\mu > 0$. Queremos encontrar

$$\begin{aligned} \sup \quad & \theta(\mu) \\ \text{s.a.} \quad & \mu \geq 0. \end{aligned}$$

Lema 1. *Considere a sequência crescente $\{\mathbf{x}_\mu\} \subset X$ sendo $\mathbf{x}_\mu$ solução do problema, então, para todo $\mu > 0$, temos que:*

$$\begin{aligned} \theta(\mu_{k+1}) &\geq \theta(\mu_k) \\ \alpha(\mathbf{x}_{\mu_{k+1}}) &\leq \alpha(\mathbf{x}_{\mu_k}) \\ f(\mathbf{x}_{\mu_{k+1}}) &\geq f(\mathbf{x}_{\mu_k}) \end{aligned}$$

Demonstração. Seja $\lambda < \mu$. Pela definição de $\theta(\lambda)$ e $\theta(\mu)$, valem as desigualdades a seguir:

$$f(\mathbf{x}_\mu) + \lambda\alpha(\mathbf{x}_\mu) \geq f(\mathbf{x}_\lambda) + \lambda f(\mathbf{x}_\lambda) \tag{1.2}$$

$$f(\mathbf{x}_\lambda) + \mu\alpha(\mathbf{x}_\lambda) \geq f(\mathbf{x}_\mu) + \mu f(\mathbf{x}_\mu) \tag{1.3}$$

visto que ambos os lados direitos são os ínfimos de expressões do mesmo tipo. Somando as desigualdades, obtemos que

$$(\mu - \lambda)\left[\alpha(\mathbf{x}_\lambda) - \alpha(\mathbf{x}_\mu)\right] \geq 0.$$

Como $\mu > \lambda$, obtemos $\alpha(\mathbf{x}_\lambda) \geq \alpha(\mathbf{x}_\mu)$. Segue então de 1.2 que $f(\mathbf{x}_\mu) \geq f(\mathbf{x}_\lambda)$ para $\lambda \geq 0$. Adicionando e subtraindo $\mu\alpha(\mathbf{x}_\mu)$ ao lado esquerdo de 1.2, obtemos

$$f(\mathbf{x}_\mu) + \mu\alpha(\mathbf{x}_\mu) + (\lambda - \mu)\alpha(\mathbf{x}_\mu) \geq \theta(\lambda).$$

Como $\mu > \lambda$ e $\alpha(\mathbf{x}_\mu) \geq 0$, a desigualdade acima implica em $\theta(\mu) \geq \theta(\lambda)$. □

Este resultado nos prepara para o seguinte teorema:

Teorema 1. *Suponha que o problema primal tenha solução viável e que para cada μ existe uma solução $\mathbf{x}_\mu \in X$ para o problema de minimizar $f(\mathbf{x}) + \mu\alpha(\mathbf{x})$ sujeita à $\mathbf{x} \in X$, e suponha ainda que $\{\mathbf{x}_\mu\}$ está contida num subconjunto compacto de X. Então*

$$\inf\{f(\mathbf{x})\colon \mathbf{g}(\mathbf{x}) \leq \mathbf{0}, \mathbf{h}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in X\} = \sup_{\mu \geq 0} \theta(\mu) = \lim_{\mu \to \infty} \theta(\mu)$$

onde $\theta(\mu) = \inf\{f(\mathbf{x}) + \mu\alpha(\mathbf{x})\colon \ \mathbf{x} \in X\} = f(\mathbf{x}_\mu) + \mu\alpha(\mathbf{x}_\mu)$. Além disso, o limite $\overline{\mathbf{x}}$ de qualquer subsequencia convergente de $\{\mathbf{x}_\mu\}$ é uma solução ótima para o problema primal e $\mu\alpha(\mathbf{x}_\mu) \to 0$ quando $\mu \to 0$.

Demonstração. Pelo lema anterior, $\theta(\mu)$ é monótona, de tal modo que $\sup_{\mu \geq 0} \theta(\mu) = \lim_{\mu \to \infty} \theta(\mu)$. Primeiro mostramos que $\alpha(\mathbf{x}_\mu) \to 0$ quando $\mu to \infty$. Seja $\mathbf{y}$ um ponto viável e $\varepsilon > 0$. Tome $\mathbf{x}_1$ como uma solução ótima ao problema de minimizar $f(\mathbf{x}) + \mu\alpha(\mathbf{x})$ sujeita à $\mathbf{x} \in X$ para $\mu = 1$. Se $\mu \geq (1/\varepsilon)\,|f(\mathbf{y}) - f(\mathbf{x}_1)| + 2$, então, pelo lema anterior, necessariamente temos que $f(\mathbf{x}_\mu) \geq f(\mathbf{x}_1)$.

Agora mostraremos que $\alpha(\mathbf{x}_\mu) \leq \varepsilon$. Por contradição, suponha que $\alpha(\mathbf{x}_\mu) > \varepsilon$. Note que, pela definição de $\theta(\mu)$,

$$\begin{aligned}\inf\{f(\mathbf{x})\colon \mathbf{g}(\mathbf{x}) \leq \mathbf{0}, \mathbf{h}(\mathbf{x}) = 0, \mathbf{x} \in X\} \geq \theta(\mu) =& f(\mathbf{x}_\mu) + \mu\alpha(\mathbf{x}_\mu) \geq f(x_1) + \mu\alpha(\mathbf{x}_1) \\ >& f(x_1) + |f(\mathbf{y}) - f(\mathbf{x}_1)| + 2\varepsilon > f(\mathbf{y}).\end{aligned}$$

A desigualdade acima não é possível em vista da viabilidade de $\mathbf{y}$. Asism, $\alpha(\mathbf{x}_\mu) \leq \varepsilon$ para todo $\mu \geq (1/\varepsilon)\,|f(\mathbf{y}) - f(\mathbf{x}_1)| + 2$. Como $\varepsilon > 0$ é arbitrário, $\alpha(\mathbf{x}_\mu) \to 0$ quando $\mu \to \infty$. Seja agora $\{\mathbf{x}_{\mu_k}\}$ uma subsequência convergente qualquer de $\{\mathbf{x}_\mu\}$ e $\overline{\mathbf{x}}$ seu limite. Então

$$\sup_{\mu \geq 0} \theta(\mu) \geq \theta(\mu) = f(\mathbf{x}_{\mu_k}) + \mu_k\alpha(\mathbf{x}_{\mu_k}) \geq f(\mathbf{x}_{\mu_k}).$$

Como $\mathbf{x}_{\mu_k} \to \overline{\mathbf{x}}$ e f é contínua, a desigualdade acima implica em

$$\sup_{\mu \geq 0} \theta(\mu) \geq f(\overline{\mathbf{x}}). \tag{1.4}$$

Como $\alpha(\mathbf{x}_\mu) \to 0$ quando $\mu \to \infty$, $\alpha(\overline{\mathbf{x}}) = 0$. Isto é, $\overline{\mathbf{x}}$ é uma solução viável do problema original. Em vista de 1.4 e da definição de $\theta(\mu)$, segue que $\overline{\mathbf{x}}$ é uma solução ótima do problema original e que $sup_{\mu \geq 0}\theta(\mu) = f(\overline{\mathbf{x}})$. Note que $\mu\alpha(\mathbf{x}_\mu) = \theta(\mu) - f(\mathbf{x}_\mu)$. Quando $\mu \to \infty$, $\theta(\mu)$ e $f(\mathbf{x}_\mu)$ ambos se aproximam de $f(\overline{\mathbf{x}})$, logo $\mu\alpha(\mathbf{x}_\mu)$ se aproxima de zero. □

A partir deste resultado fica claro que podemos nos aproximar arbitrariamente do valor ótimo do problema primal ao computar $\theta(\mu)$ para μ suficientemente grande, pois assim $f(\mathbf{x}_\mu) + \mu\alpha(\mathbf{x}_\mu)$ se aproxima do ótimo primal. Segue também para μ suficientemente grande que $\mathbf{x}_\mu$ pode ser arbitrariamente aproximado da região viável. Este resultado sugere o método de resolver uma sequência de problemas da forma:

$$\begin{aligned}\text{minimizar} \quad & f(\mathbf{x}) + \mu\alpha(\mathbf{x}) \\ \text{s.a.} \quad & \mathbf{x} \in X\end{aligned}$$

para uma sequência crescente de μ. Assim, esse esquema é como se estivéssemos nos aproximando do ponto ótimo a partir da região inviável.

1.3 Dificuldades computacionais

Seja $F(\mathbf{x}) = f(\mathbf{x}) + \mu \sum_{i=1}^{l} \psi[h_i(\mathbf{x})]$. Estudando

$$\nabla F(\mathbf{x}) = \nabla f(\mathbf{x}) + \mu \sum_{i=1}^{l} \psi' \left[h_i(\mathbf{x})\right] \nabla h_i(\mathbf{x})$$

$$\nabla^2 F(\mathbf{x}) = \left[\nabla^2 f(\mathbf{x}) + \sum_{i=1}^{l} \mu\psi' \left[h_i(\mathbf{x})\right] \nabla^2 h_i(\mathbf{x})\right] + \mu \sum_{i=1}^{l} \psi'' \left[h_i(\mathbf{x})\right] \nabla h_i(\mathbf{x}) \nabla h_i(\mathbf{x})^T,$$

encontramos que o termo esquerdo se aproxima de um valor finito, porém o termo direito é potencialmente explosivo. Assim, a matriz hessiana está severamente mal-condicionada, o que pode ser desastroso para o método do gradiente. Como o método do gradiente utiliza uma aproximação linear e escolhe uma direção $-\nabla f(\mathbf{x})$, as direções subsequentes são aproximadamente ortogonais e perto do ótimo perdem a qualidade devido a ignorar-se os termos não-lineares, de modo que o método *ziguezagueia* na região próxima. Vamos definir o *número de condicionamento* de uma matriz definida positiva como sendo a razão do seu maior autovalor para o menor. Pode ser mostrado que, à medida que o número de condicionamento de $\nabla^2 f(\mathbf{x})$ aumenta, os contornos de $f(\mathbf{x})$ se tornam cada vez mais densos ao longo de certas direções, fenômeno conhecido como *mau-condicionamento*. Isso é desastroso para o método do gradiente, principalmente levando em conta a natureza explosiva do segundo termo da equação acima.

Outros métodos como Newton, gradientes conjugados e quase-Newton não são suscetíveis ao mesmo fenômeno, pois adotam um passo da forma $-D\nabla f(\mathbf{x})$, "defletindo" a direção do gradiente. Portanto, o método básico do gradiente não será implementado, serão implementados apenas os outros três métodos vistos na disciplina.

Além disso, se escolhermos μ muito grande e tentarmos resolver o problema de penalidade, será dada ênfase à viabilidade e pode acontecer terminação prematura, com um ótimo distante do ótimo primal. Na presença de restrições não lineares, movimentos na direção $\mathbf{d}$ a partir de $\mathbf{x}$ podem levar a pontos não viáveis ou a um grande aumento no valor de função auxiliar, ou seja, $f(\mathbf{x}+\lambda\mathbf{d}) + \mu\alpha(\mathbf{x}+\lambda\mathbf{d}) > f(\mathbf{x}) + \mu\alpha(\mathbf{x})$. Uma melhora só é possível caso o tamanho do passo λ seja bastante pequeno, o que também pode levar à terminação prematura e convergência lenta.

1.4 Método e algoritmo

Para combater esses problemas, vamos utilizar a abordagem sugerida pelo teorema 1 de resolver uma sequência de problemas de penalidade com valores crescentes de μ, cada um iniciando do ótimo encontrado para o problema anterior. Segue um resumo do método empregado:

Inicialização Seja $\varepsilon > 0$ um valor de terminação, $\mathbf{x}_1$ um ponto inicial, $\mu_1 > 0$ um parâmetro de penalidade, $\beta > 1$ e $k = 1$.

Passo principal

1. Começando a partir de $\mathbf{x}_k$, solucione o seguinte problema:

$$\begin{array}{rl} \text{minimizar} & f(\mathbf{x}) + \mu_k \alpha(\mathbf{x}) \\ \text{s.a.} & \mathbf{x} \in X. \end{array}$$

   Tome o ótimo como $\mathbf{x}_{k+1}$ prossiga para o passo seguinte.

2. Se $\mu_k \alpha(\mathbf{x}_{k+1}) < \varepsilon$, termine. Caso contrário, faça $\mu_{k+1} = \beta \mu_k$, $k = k + 1$ e retome o passo anterior.

2 Método da barreira

2.1 Funções de barreira

O método consiste em gerar uma sequência de pontos viáveis que convergirá para a solução. Para tanto, o problema de otimização restrita será transformado em um irrestrito, da mesma forma que o problema da penalidade. Desse modo, a solução do problema de otimização restrito é aproximado por uma sequência de problemas irrestritos. A ideia é iniciar o método no interior da região viável e impor uma barreira para se deixar essa região.

Seja $f\colon \mathbb{R}^n \to \mathbb{R}$ e $g\colon \mathbb{R}^n \to \mathbb{R}^m$, o problema a ser resolvido é o seguinte:

Problema primal

$$\begin{aligned} \text{minimizar} \quad & f(\mathbf{x}) \\ \text{s.a.} \quad & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \\ & \mathbf{x} \in X \end{aligned}$$

Problema de barreira

Seja B uma função não negativa e contínua em $\{\mathbf{x}\colon \mathbf{g}(\mathbf{x}) < 0\}$ e $\theta(\mu) = \inf\{f(\mathbf{x}) + \mu B(\mathbf{x})\colon \ \mathbf{g}(\mathbf{x}) < 0, \mathbf{x} \in X\}$. Queremos encontrar

$$\begin{aligned} \inf \quad & \theta(\mu) \\ \text{s.a.} \quad & \mu \geq 0. \end{aligned}$$

Note que, para esse método, é necessário que o conjunto $\{x\colon \mathbf{g}(\mathbf{x}) < 0\}$ seja não-vazio, o que não seria possível se as restrições de igualdade fossem acomodadas dentro do conjunto de desigualdades. Qualquer restrição de igualdade deve ser incorporada ao conjunto X ou ser eliminada através de redução da dimensionalidade do problema.

Uma função barreira é definida por

$$B(\mathbf{x}) = \sum_{i=1}^{m} \phi[g_i(\mathbf{x})], \tag{2.1}$$

onde ϕ é contínua em $\{y\colon y < 0\}$ e satisfaz

$$\phi(y) \geq 0 \text{ se } y < 0 \quad \text{e} \quad \lim_{y \to 0^-} \phi(y) = \infty$$

Considere $\bar{X}$ o interior do conjunto X. A barreira do conjunto X é a função $B\colon \bar{X} \to \mathbb{R}^+$. Tipicamente, escolhemos a barreira logarítmica,

$$B = -\sum_{n=1}^{m} \log(-g_i(x)), \tag{2.2}$$

ou barreira inversa

$$B = -\sum_{n=1}^{m} \frac{1}{g_i(x)}. \tag{2.3}$$

Lema 2. *Considere a sequência* $\{\mathbf{x}_k\} \subset X$ *tal que* $\mathbf{g}(\mathbf{x}_k) < 0$ *e* $f(\mathbf{x}_k) + \mu B(\mathbf{x}_k) \to \theta(\mu)$, *de modo que* $\{\mathbf{x}_k\}$ *possui subsequência convergente. Vale:*

1. *Para cada* $\mu > 0$, *existe algum* $\mathbf{x}_\mu \in X$ *com* $\mathbf{g}(\mathbf{x}_\mu) < \mathbf{0}$ *tal que* $\theta(\mu) = f(\mathbf{x}_\mu) + \mu B(\mathbf{x}_\mu) = \inf\{f(\mathbf{x}) + \mu B(\mathbf{x})\colon \mathbf{g}(\mathbf{x}) < 0, \mathbf{x} \in X\}$.
2. $\inf\{f(\mathbf{x})\colon \mathbf{g}(\mathbf{x}) \leq \mathbf{0}, \mathbf{x} \in X\} \leq \inf\{\theta(\mu)\colon \mu > 0\}$.
3. *a)* $\theta(\mu_{k+1}) \leq \theta(\mu_k)$
   b) $B(\mathbf{x}_k) \leq B(\mathbf{x}_{k+1})$
   c) $f(\mathbf{x}_{k+1}) \leq f(\mathbf{x}_k)$.

Demonstração. Fixe $\mu > 0$. Pela definição de θ, existe uma sequência $\{\mathbf{x}_k\}$ com $\mathbf{x}_k \in X$ e $\mathbf{g}(\mathbf{x}_k) < \mathbf{0}$ tal que $f(\mathbf{x}_k) + \mu B(\mathbf{x}_k) \to \theta(\mu)$. Por hipótese, $\{\mathbf{x}_k\}$ tem uma subsequência convergente $\{\mathbf{x}_k\}_K$ com limite $\mathbf{x}_\mu \in X$. Pela continuidade de $\mathbf{g}$, $\mathbf{g}(\mathbf{x}_\mu) \leq \mathbf{0}$. Mostraremos que $\mathbf{g}(\mathbf{x}_\mu) < \mathbf{0}$. Caso contrário, $g_i(\mathbf{x}_\mu) = 0$ para algum i, e como a função barreira B satisfaz 2.1, para $k \in K$, $B(\mathbf{x}_k) \to \infty$. Logo, $\theta(\mu) = \infty$, o que é impossível, visto que $\{\mathbf{x}\colon \mathbf{x} \in X, g(\mathbf{x}) < 0\}$ é assumido não-vazio. Portanto, $\theta(\mu) = f(\mathbf{x}_\mu) + \mu B(\mathbf{x}_\mu)$, onde $\mathbf{x}_\mu \in X$ e $\mathbf{g}(\mathbf{x}_\mu) < 0$, de modo que a parte 1 está demonstrada. Agora, como $B(\mathbf{x}) \geq 0$ se $\mathbf{g}(\mathbf{x}) < \mathbf{0}$, então, para $\mu \geq 0$, temos

$$\begin{aligned}
\theta(\mu) &= \inf\{f(\mathbf{x}) + \mu B(\mathbf{x})\colon \mathbf{g}(\mathbf{x}) < \mathbf{0}, \mathbf{x} \in X\} \\
&\geq \inf\{f(\mathbf{x})\colon \mathbf{g}(x) < \mathbf{0}, \mathbf{x} \in X\} \\
&\geq \inf\{f(\mathbf{x})\colon \mathbf{g}(x) \leq \mathbf{0}, \mathbf{x} \in X\}.
\end{aligned}$$

Como a desigualdade acima vale para cada $\mu \geq 0$, a parte 2 segue. Para mostrar a parte 3, considere $\mu > \lambda > 0$. Como $B(\mathbf{x}) \geq 0$ se $\mathbf{g}(\mathbf{x}) < 0$, então $f(\mathbf{x}) + \mu B(\mathbf{x}) \geq f(\mathbf{x}) + \lambda B(\mathbf{x})$ para cada $\mathbf{x} \in X$ com $\mathbf{g}(\mathbf{x}) < 0$. Logo, $\theta(\mu) \geq \theta(\lambda)$. Notando a parte 1, existem $\mathbf{x}_\mu$ e $\mathbf{x}_\lambda$ tais que

$$f(\mathbf{x}_\mu) + \mu B(\mathbf{x}_\mu) \leq f(\mathbf{x}_\lambda) + \mu B(\mathbf{x}_\lambda) \tag{2.4}$$

$$f(\mathbf{x}_\lambda) + \lambda B(\mathbf{x}_\lambda) \leq f(\mathbf{x}_\mu) + \lambda f(\mathbf{x}_\mu). \tag{2.5}$$

Adicionando as duas desigualdades e rearranjando, obtemos $(\mu - \lambda)\,[B(\mathbf{x}_\mu) - B(\mathbf{x}_\lambda)] \leq 0$. Como $\mu - \lambda > 0$, então $B(\mathbf{x}_\mu) \leq B(\mathbf{x}_\lambda)$. Substituindo em 2.1, segue que $f(\mathbf{x}_\lambda) \leq f(\mathbf{x}_\mu)$. Assim, a parte 3 está demonstrada. □

Este resultado nos prepara para o teorema 2.

Teorema 2. *Suponha que o problema primal tenha solução ótima* $\overline{\mathbf{x}}$ *com a seguinte propriedade. Dada qualquer vizinhança* N *em torno de* $\overline{\mathbf{x}}$, *existe* $\overline{x} \in X \cap N$ *tal que* $\mathbf{g}(\mathbf{x}) < 0$. *Então, para* $\mathbf{x} \in X$ *e* $\mathbf{g}(\mathbf{x}) < 0$,

$$\min\{f(\mathbf{x})\colon \mathbf{g}(\mathbf{x}) \leq \mathbf{0},\, \mathbf{x} \in X\} = \lim_{\mu \to 0^+} \theta(\mu) = \inf_{\mu > 0} \theta(\mu).$$

Fazendo $\theta(\mu) = f(\mathbf{x}_\mu) + \mu B(\mathbf{x}_\mu)$, o limite de qualquer subsequência convergente de $\{\mathbf{x}_k\}$ é uma solução ótima do problema primal, além disso, $\mu B(\mathbf{x}_\mu) \to 0$ quando $\mu \to 0^-$.

Demonstração. Seja $\overline{\mathbf{x}}$ uma solução ótima do problema primal satisfazendo a propriedade mencionada, e seja $\varepsilon > 0$. Pela continuidade de f e pela suposição do teorema, existe $\hat{\mathbf{x}} \in X$ com $\mathbf{g}(\hat{\mathbf{x}}) < \mathbf{0}$ tal que $f(\overline{\mathbf{x}}) + \varepsilon > f(\hat{\mathbf{x}})$. Então, para $\mu > 0$,

$$f(\overline{\mathbf{x}}) + \varepsilon + \mu B(\hat{\mathbf{x}}) > f(\hat{\mathbf{x}}) + \mu B(\hat{\mathbf{x}}) \geq \theta(\mu).$$

Tomando o limite em que $\mu \to 0^+$, segue que $f(\overline{\mathbf{x}}) + \varepsilon \geq \lim_{\mu \to 0^+} \theta(\mu)$. Como essa desigualdade continua verdadeira para cada $\varepsilon > 0$, obtemos $f(\overline{\mathbf{x}}) \geq \lim_{\mu \to 0^+} \theta(\mu)$. Em vista da parte 2 do lema anterior, $f(\overline{\mathbf{x}}) = \lim_{\mu \to 0^+} \theta(\mu)$.

Para $\mu \to 0^+$ e como $B(\mathbf{x}_\mu) \geq 0$ e $\mathbf{x}_\mu$ é viável para o problema original, segue que

$$\theta(\mu) = f(\mathbf{x}_\mu) + \mu B(\mathbf{x}_\mu) \geq f(\mathbf{x}_\mu) \geq f(\overline{\mathbf{x}}).$$

Tomando o limite em que $\mu \to 0^+$ e notando que $f(\overline{\mathbf{x}}) = \lim_{\mu \to 0^+} \theta(\mu)$, segue que ambos $f(\mathbf{x}_\mu)$ e $f(\mathbf{x}_\mu) + \mu B(\mathbf{x}_\mu)$ se aproximam de $f(\overline{\mathbf{x}})$. Logo, $\mu B(\mathbf{x}_\mu) \to 0$ quando $\mu \to 0^+$. Além disso, se $\{\mathbf{x}_\mu\}$ tem uma subsequência convergent com limite $\mathbf{x}'$, então $f(\mathbf{x}') = f(\overline{\mathbf{x}})$. Como $\mathbf{x}_\mu$ é viável para o problema original para cada μ, segue que $\mathbf{x}'$ também é viável e, portanto, ótimo. □

2.2 Método e algoritmo

Em face do teorema 2 acima e da discussão anterior, notamos que o método da barreira é semelhante ao da penalidade, porém com algumas ideias contrárias. Vamos resolver uma sequência de problemas de barreira de modo a nos aproximarmos de uma solução do problema primal. No entanto, o nosso parâmetro de barreira μ decresce enquanto que a função barreira $B(\mathbf{x})$ cresce a cada iteração.

Este método sofre dos mesmos problemas de mau-condicionamento que o problema da penalidade. Além disso, o problema da barreira necessita incorporar a restrição $\mathbf{g}(\mathbf{x}) < 0$. Embora comecemos com um ponto interior e a barreira $B(\mathbf{x})$ tenda a infinito ao nos aproximarmos da fronteira de $\{\mathbf{x} \colon \mathbf{g}(\mathbf{x}) < 0\}$, os métodos de busca linear utilizados dão passos discretos e podem violar as condições de viabilidade, levando a uma região onde $B(\mathbf{x}^*)$ assume um valor negativo muito alto. Dessa forma, a cada cálculo do tamanho do passo λ, $\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda \mathbf{d}$, verificamos a viabilidade do próximo ponto $\mathbf{x}_{k+1}$.

Inicialização Seja $\varepsilon > 0$ um valor de terminação, $\mathbf{x}_1 \in X$ com $\mathbf{g}(\mathbf{x}_1) < 0$, $\mu > 0$, $\beta \in (0, 1)$ e $k = 1$.

Passo principal

1. Começando de $\mathbf{x}_k$, resolva o seguinte problema:

$$\begin{array}{rl} \text{minimizar} & f(\mathbf{x}) + \mu B(\mathbf{x}) \\ \text{s.a.} & \mathbf{g}(\mathbf{x}) < \mathbf{0} \\ & \mathbf{x} \in X \end{array}$$

Tome o ótimo como $\mathbf{x}_{k+1}$ e prossiga para o passo seguinte.

2. Se $\mu_k B(\mathbf{x}_{k+1}) < \varepsilon$, termine. Caso contrário, faça $\mu_{k+1} = \beta\mu_k$, $k = k + 1$ e retome o passo anterior.

3 Resultados

Onde não for indicado diferente, em todos os testes de barreira utilizamos barreira logarítmica (equação 2.2), com erro $\varepsilon = 10^{-6}$ e precisão $\delta = 10^{-6}$. Também escolhemos $\mu = 10$ e $\beta = 0.1$. Em todos os testes de penalidade, utilizamos a função de penalidade $\alpha(\mathbf{x})$ como definida em 1.1 com potência $p = 2$, com os mesmos valores e erro e precisão. Nesse caso, o parâmetro $\mu = 0.1$ e $\beta = 10$. Para a busca de Armijo, onde aplicável escolhemos $\eta = 0.5$ e, no método de quase-Newton, o parâmetro $\sigma = 1$ (onde $H_1 = \sigma I$). No método de Newton o parâmetro $\sigma = 0$, pois valores positivos não são recomendados (queremos evitar nos aproximarmos do método do gradiente).

3.1 Método de barreira

Os três problemas seguintes serão solucionados pelo Método de Barreira:

3.1.1 Primeiro problema teste

$$\begin{aligned} \min \quad & f_{11}(\mathbf{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \\ \text{s.a} \quad & x_1 - x_2^2 \leq 0 \\ & x_1^2 - x_2 \geq 0 \\ & \mathbf{x}_0 = [-2 \;\; 1]^t. \end{aligned}$$

Partindo do ponto inicial $\mathbf{x}_0$, primeiramente testamos a função na forma irrestrita, para depois executá-la na forma restrita. Ao executar o teste, utilizamos os seguintes parâmetros: método irrestrito: gradientes conjugados; método de busca linear: seção-áurea;

```
9: mu: 10, B(x): -0, mu*B(x): -0,
0: { -2, 1 }
1: { 1.35496, 1.83665 }
2: { 1.24708, 1.54393 }
3: { 1.11274, 1.22882 }
4: { 1.02481, 1.05416 }
5: { 0.986907, 0.975177 }
6: { 1.00056, 1.00119 }
7: { 0.999998, 0.999995 }
8: { 1, 1 }
x_final = { 1, 1 }
```

```
f_final = 1.71393e-15
steps = 9
||grad(x)||: 1.2725e-06
g(x) < 0: true
```

Perceba que executando o problema na forma irrestrita, tem-se uma trajetória com 9 passos, onde cada ponto segue uma "sequência", ou seja, são bem próximos um do outro, até se chegar na solução ótima. Já para na forma restrita, foram obtidos 80 passos, onde os pontos não foram se aproximando monotonicamente da solução ótima. Veja abaixo alguns pontos que mostram esta quebra nas sequências graças ao fato que os métodos restritos resolvem uma sequência de problemas com funções objetivos diferentes (μ cada vez maior), de modo que o avanço fica inconsistente. Além disso, o método tenta se afastar da região inviável e se manter no interior do conjunto X.

Ao executar o teste, utilizamos os seguintes parâmetros: método irrestrito: gradientes conjugados; método de busca linear: seção-áurea;

```
16: mu: 10, B(x): -4.93168, mu*B(x): -49.3168,
63: mu: 1, B(x): 3.5589, mu*B(x): 3.5589,
73: mu: 0.1, B(x): 10.0869, mu*B(x): 1.00869,
74: mu: 0.01, B(x): 10.0869, mu*B(x): 0.100869,
75: mu: 0.001, B(x): 10.0869, mu*B(x): 0.0100869,
76: mu: 0.0001, B(x): 10.0869, mu*B(x): 0.00100869,
77: mu: 1e-05, B(x): 10.0869, mu*B(x): 0.000100869,
78: mu: 1e-06, B(x): 10.0869, mu*B(x): 1.00869e-05,
79: mu: 1e-07, B(x): 10.0869, mu*B(x): 1.00869e-06,
80: mu: 1e-08, B(x): 10.0869, mu*B(x): 1.00869e-07,

x_final = { 6.70318, -2.58905 }
f_final = 225863
steps = 80
||grad(x)||: 127784
```

Para demonstrar a importância da escolha dos submétodos, repetimos o teste utilizando diferentes métodos de busca linear. Para o método de Newton, obtivemos o seguinte resultado:

```
23: mu: 10, B(x): -4.93167, mu*B(x): -49.3167,
40: mu: 1, B(x): 3.31516, mu*B(x): 3.31516,
43: mu: 0.1, B(x): 3.42778, mu*B(x): 0.342778,
49: mu: 0.01, B(x): 3.77674, mu*B(x): 0.0377674,
50: mu: 0.001, B(x): 3.77674, mu*B(x): 0.00377674,
51: mu: 0.0001, B(x): 3.77674, mu*B(x): 0.000377674,
52: mu: 1e-05, B(x): 3.77674, mu*B(x): 3.77674e-05,
53: mu: 1e-06, B(x): 3.77674, mu*B(x): 3.77674e-06,
54: mu: 1e-07, B(x): 3.77674, mu*B(x): 3.77674e-07,
```

```
x_final = { -0.446046, 0.150092 }
f_final = 2.32983
steps = 54
||grad(x)||: 15.1763
g(x) < 0: true
```

Para a busca de Armijo, obtivemos o seguinte resultado:

```
4: mu: 10, B(x): -0.609641, mu*B(x): -6.09641,
15: mu: 1, B(x): 3.32406, mu*B(x): 3.32406,
22: mu: 0.1, B(x): 6.77401, mu*B(x): 0.677401,
32: mu: 0.01, B(x): 10.2451, mu*B(x): 0.102451,
37: mu: 0.001, B(x): 13.6897, mu*B(x): 0.0136897,
44: mu: 0.0001, B(x): 17.1642, mu*B(x): 0.00171642,
49: mu: 1e-05, B(x): 20.5721, mu*B(x): 0.000205721,
51: mu: 1e-06, B(x): 22.4967, mu*B(x): 2.24967e-05,
52: mu: 1e-07, B(x): 22.4967, mu*B(x): 2.24967e-06,
53: mu: 1e-08, B(x): 22.4967, mu*B(x): 2.24967e-07,

x_final = { -6.83408e-07, -0.000230474 }
f_final = 1.00001
steps = 53
||grad(x)||: 2.00053
g(x) < 0: true
```

Observe que nos dois primeiros testes o ponto estacionário é atingido mais cedo na sequência de problemas irrestritos, que não conseguem melhorar o ótimo "parcial" encontrado. O laço principal da função minimize_barrier_method escala o parâmetro μ e tenta solucionar um novo subproblema, porém o submétodo irrestrito não consegue melhorar o ponto numa direção viável a partir do primeiro passo, de modo que o laço principal apenas diminui o parâmetro μ até que o critério de terminação $\mu B(\mathbf{x}_k) < 10^{-6}$ seja satisfeito.

Já utilizando a busca linear de Armijo, fomos capazes de seguir outra trajetória e uma quantidade maior de subproblemas foram capazes de aprimorar o ótimo parcial. Isso levou a um valor final de $f(\overline{\mathbf{x}})$ e $\nabla f(\overline{\mathbf{x}})$.

No entanto, podemos obter um resultado muito similar ao de acima com o método da seção-áurea para busca linear se utilizarmos o método de Newton para resolver os subproblemas irrestritos:

```
4: mu: 10, B(x): -1.13083, mu*B(x): -11.3083,
8: mu: 1, B(x): -0.111013, mu*B(x): -0.111013,
11: mu: 0.1, B(x): 0.391864, mu*B(x): 0.0391864,
15: mu: 0.01, B(x): -0.351021, mu*B(x): -0.00351021,
19: mu: 0.001, B(x): -2.12168, mu*B(x): -0.00212168,
33: mu: 0.0001, B(x): 17.1532, mu*B(x): 0.00171532,
38: mu: 1e-05, B(x): 20.3965, mu*B(x): 0.000203965,
```

```
40: mu: 1e-06, B(x): 23.9767, mu*B(x): 2.39767e-05,
41: mu: 1e-07, B(x): 23.9767, mu*B(x): 2.39767e-06,
42: mu: 1e-08, B(x): 23.9767, mu*B(x): 2.39767e-07,

x_final = { -7.63794e-08, -0.000264265 }
f_final = 1.00001
steps = 42
||grad(x)||: 2.0007
g(x) < 0: true
```

Ressaltando que todas essas significativas diferenças nos resultados foram obtidas sem manipular os parâmetros específicos de cada submétodo (ex: ρ, η, δ).

3.1.2 Segundo problema teste

$$
\begin{aligned}
\min \quad & f_{12}(\mathbf{x}) = x_1^2 + 0,5x_2^2 + x_3^2 + 0,5x_4^2 - x_1x_3 + x_3x_4 - x_1 - 3x_2 + x_3 - x_4 \\
\text{s.a} \quad & 5 - x_1 - 2x_2 - x_3 - x_4 \geq 0 \\
& 4 - 3x_1 - x_2 - 2x_3 + x_4 \geq 0 \\
& x_2 + 4x_3 - 1,5 \geq 0 \\
& x_i \geq 0, \quad \forall i \in \{1, 2, 3, 4\} \\
& \mathbf{x}^0 = [0,5 \;\; 0,5 \;\; 0,5 \;\; 0,5]^t.
\end{aligned}
$$

O teste 2 parece ser dotado de alguma propriedade que o torna bastante computacionalmente intensivo para os métodos de busca linear da forma que implementamos, a não para o método de Armijo. Sabemos disso pois testamos o método na forma irrestrita.

Ao executar o teste, utilizamos os seguintes parâmetros: método irrestrito: gradientes conjugados; método de busca linear: Armijo;

```
30: mu: 10, B(x): -0, mu*B(x): -0,
0: { 0.5, 0.5, 0.5, 0.5 }
1: { 0.756, 1.78, -0.524, 0.5 }
2: { 0.159304, 2.89252, -0.594817, 1.15536 }
3: { 0.0795617, 3.31009, -1.42098, 1.77749 }
...
26: { -0.999997, 3, -2.99999, 4 }
27: { -0.999998, 3, -3, 4 }
28: { -0.999999, 3, -3, 4 }
29: { -0.999999, 3, -3, 4 }
x_final = { -0.999999, 3, -3, 4 }
f_final = -7.5
steps = 30
||grad(x)||: 7.793e-07
g(x) < 0: true
```

Solucionando o problema restrito:

```
19: mu: 10, B(x): -1.71283, mu*B(x): -17.1283,
41: mu: 1, B(x): -0.386127, mu*B(x): -0.386127,
70: mu: 0.1, B(x): 3.96031, mu*B(x): 0.396031,
101: mu: 0.01, B(x): 8.61528, mu*B(x): 0.0861528,
148: mu: 0.001, B(x): 13.2281, mu*B(x): 0.0132281,
154: mu: 0.0001, B(x): 17.8527, mu*B(x): 0.00178527,
159: mu: 1e-05, B(x): 23.3247, mu*B(x): 0.000233247,
161: mu: 1e-06, B(x): 25.819, mu*B(x): 2.5819e-05,
162: mu: 1e-07, B(x): 25.819, mu*B(x): 2.5819e-06,
163: mu: 1e-08, B(x): 25.819, mu*B(x): 2.5819e-07,
0: { 0.5, 0.5, 0.5, 0.5 }
1: { 0.467718, 0.664483, 0.506149, 1.078 }
2: { 0.587558, 0.519759, 0.558475, 1.1756 }
3: { 0.536437, 0.575293, 0.545984, 1.92908 }
4: { 0.480979, 0.510497, 0.490531, 2.06124 }
...
159: { 0.273686, 2.08923, 5.00904e-06, 0.547846 }
160: { 0.273688, 2.08923, 6.37358e-07, 0.547848 }
161: { 0.273688, 2.08923, 6.37358e-07, 0.547848 }
162: { 0.273688, 2.08923, 6.37358e-07, 0.547848 }
x_final = { 0.273688, 2.08923, 6.37358e-07, 0.547848 }
f_final = -4.68181
steps = 163
||grad(x)||: 1.69183
g(x) < 0: true
```

É interessante observar, mais um vez, o fenômeno de que a sequência final de problemas irrestritos não consegue melhorar o ótimo logo de início (nos passos $159, 161, 162, 163$). É uma indicação visível de que a trajetória "atingiu a barreira".

3.1.3 Terceiro problema teste

$$\begin{aligned} \min \quad & f_{13}(\mathbf{x}) = (x_1 - 10)^3 + (x_2 - 20)^3 \\ \text{s.a} \quad & (x_1 - 5)^2 + (x_2 - 5)^2 - 100 \geq 0 \\ & (x_1 - 6)^2 + (x_2 - 5)^2 \geq 0 \\ & 82,81 - (x_1 - 6)^2 - (x_2 - 5)^2 \geq 0 \\ & \mathbf{x} = [14,115 \;\; 0,885]^t. \end{aligned}$$

Primeiramente, vamos encontrar um ótimo global. Ao executar o teste, utilizamos os seguintes parâmetros: método irrestrito: Newton; método de busca linear: Armijo;

```
23: mu: 10, B(x): -0, mu*B(x): -0,
```

```
0: { 13, 0 }
1: { 11.4969, 9.99556 }
2: { 10.7508, 14.9978 }
3: { 10.3754, 17.4989 }
4: { 10.1877, 18.7494 }
5: { 10.0938, 19.3747 }
6: { 10.0469, 19.6874 }
7: { 10.0235, 19.8437 }
10: { 10.0029, 19.9805 }
16: { 10, 19.9997 }
17: { 10, 19.9998 }
18: { 10, 19.9999 }
22: { 10, 20 }
x_final = { 10, 20 }
f_final = -1.06814e-16
steps = 23
||grad(x)||: 6.86349e-11
g(x) < 0: true
```

Da mesma forma que o problema teste 1, tem-se que na forma irrestrita a busca pela solução é rápida, mesmo o ponto inicial não estando tão perto do ponto ótimo quanto no problema 1. Ao Aplicar-se a busca pelo Método de Barreira com este ponto inicial, chegamos ao resultado de que o ponto inicial é inviável, impossibilitando-nos assim de encontrar o próximo ponto.

Ao executar o teste, utilizamos os seguintes parâmetros: método irrestrito: Newton; método de busca linear: Armijo;

```
Error: infeasible starting point.
```

Uma análise gráfica das restrições, corroborada por uma análise algébrica, mostra que o conjunto viável é bastante "estreito", podendo ser desafiador para métodos restritos dependendo da capacidade de cada implementação de lidar com níveis de precisão. Também observamos que a segunda restrição é inócua, pois se trata da região exterior de um círculo de raio zero. A região viável encontra-se em:

$$\begin{aligned} 14.095 < & x_1 < 15 \\ (1/10)(50 - \sqrt{-100x^2 + 1200x + 4681}) \leq & x_2 \leq 5 - \sqrt{-x^2 + 10x + 75} \end{aligned}$$

Portanto, testaremos o problema na forma restrita com $\mathbf{x}_0 = [14.115, 0.885]^T$. Ao executar o teste, utilizamos os seguintes parâmetros: método irrestrito: Newton; método de busca linear: Armijo;

```
2: mu: 10, B(x): -2.60429, mu*B(x): -26.0429,
4: mu: 1, B(x): -2.60543, mu*B(x): -2.60543,
6: mu: 0.1, B(x): -2.60641, mu*B(x): -0.260641,
```

```
8: mu: 0.01, B(x): -2.60652, mu*B(x): -0.0260652,
10: mu: 0.001, B(x): -2.60519, mu*B(x): -0.00260519,
12: mu: 0.0001, B(x): -2.60302, mu*B(x): -0.000260302,
14: mu: 1e-05, B(x): -2.60023, mu*B(x): -2.60023e-05,
16: mu: 1e-06, B(x): -2.59696, mu*B(x): -2.59696e-06,
18: mu: 1e-07, B(x): -2.59318, mu*B(x): -2.59318e-07,
20: mu: 1e-08, B(x): -2.58871, mu*B(x): -2.58871e-08,
22: mu: 1e-09, B(x): -2.58369, mu*B(x): -2.58369e-09,
24: mu: 1e-10, B(x): -2.57665, mu*B(x): -2.57665e-10,
26: mu: 1e-11, B(x): -2.5687, mu*B(x): -2.5687e-11,
0: { 14.5, 1.81394 }
1: { 14.5001, 1.81425 }
2: { 14.5001, 1.81425 }
...
24: { 14.4994, 1.82131 }
25: { 14.4992, 1.82208 }
x_final = { 14.4992, 1.82208 }
f_final = -5915.58
steps = 26
||grad(x)||: 993.169
g(x) < 0: true
```

Diferentemente do problema de teste 1, o Método de Barreira não diversificou tanto a busca por pontos viáveis no espaço solução, desta vez houve uma maior intensificação na região inicial, onde muito provavelmente seja pelo ponto inicial já estar perto da barreira.

3.2 Método de penalidade

A execução dos testes ocorrerá da mesma forma dos problemas anteriores, primeiro executaremos na forma irrestrita e logo após na forma restrita.

3.2.1 Primeiro problema teste

$$\begin{aligned} \min \quad & f_{21}(\mathbf{x}) = 0.01(x_1 - 1)^2 + (x_2 - x_1^2)^2 \\ \text{s.a} \quad & x_1 + x_3^2 + 1 = 0 \\ & \mathbf{x}^0 = [2 \;\; 2 \;\; 2]^t. \end{aligned}$$

Ao executar o teste irrestrito, utilizamos os seguintes parâmetros: método irrestrito: gradientes conjugados; método de busca linear: Armijo;

```
19: mu: 0.1, a(x): 0, mu*a(x): 0
0: { 2, 2, 2 }
1: { 1.54908, 2.11259, 2 }
```

```
2: { 1.5852, 2.23082, 2 }
3: { 1.53556, 2.35227, 2 }
4: { 1.51741, 2.29644, 2 }
5: { 1.48883, 2.22547, 2 }
6: { 1.32235, 1.73235, 2 }
7: { 1.20852, 1.44656, 2 }
8: { 1.16373, 1.35643, 2 }
9: { 1.11008, 1.23057, 2 }
10: { 1.01968, 1.03953, 2 }
11: { 1.0123, 1.02553, 2 }
12: { 1.00042, 1.00067, 2 }
13: { 0.999505, 0.998947, 2 }
14: { 0.999988, 0.999956, 2 }
15: { 0.999902, 0.999794, 2 }
16: { 0.999997, 0.999992, 2 }
17: { 0.999991, 0.999982, 2 }
18: { 1, 1, 2 }
x_final = { 1, 1, 2 }
f_final = 1.67237e-14
steps = 19
||grad(x)||: 5.76049e-07
g(x) < 0: true
```

E obtemos os seguintes resultados na forma restrita:

```
14: mu: 0.1, a(x): 0.178496, mu*a(x): 0.0178496
19: mu: 1, a(x): 0.00469089, mu*a(x): 0.00469089
53: mu: 10, a(x): 3.9919e-06, mu*a(x): 3.9919e-05
58: mu: 100, a(x): 4.02353e-08, mu*a(x): 4.02353e-06
65: mu: 1000, a(x): 4.47379e-10, mu*a(x): 4.47379e-07
0: { 2, 2, 2 }
1: { 1.50967, 2.11259, 1.84237 }
2: { 1.59541, 2.04799, 0.914463 }
...
15: { -0.707176, 0.32107, -0.0275771 }
16: { -0.855875, 0.684823, 0.0134995 }
17: { -0.870173, 0.688248, 0.00901943 }
...
32: { -0.998021, 0.996129, -0.00233816 }
33: { -0.998041, 0.996093, -0.00225402 }
34: { -0.997972, 0.995816, -0.00157142 }
...
62: { -0.999974, 0.999948, -2.83032e-05 }
63: { -0.999979, 0.999948, -2.8302e-05 }
64: { -0.999979, 0.999958, -2.73612e-05 }
```

```
x_final = { -0.999979, 0.999958, -2.73612e-05 }
f_final = 0.0399992
steps = 65
||grad(x)||: 0.0399991
g(x) < 0: true
```

O método de penalidade foi aplicado com sucesso com estes métodos de otimização e de busca linear, os pontos vieram decaindo a cada passo, no entanto ao manipular-se a função de execução alterando o método de busca linear para newton, obtém-se um tempo de compilação muito maior que quando utilizado o método de Armijo. Este problema de teste não foi corretamente solucionado utilizando os métodos de otimização irrestritos que não o método de gradientes conjugados com a busca linear de Armijo na maioria das vezes.

Isto indica que **ainda existem fraquezas que ainda não foram corrigidas na nossa implementação.** Entretanto, a combinação de **gradientes conjugados** para o método irrestrito e **busca de Armijo** para busca linear se mostrou estável e consistente para todos os testes.

3.2.2 Segundo problema teste

Este problema teste foi executado utilizando o gradiente conjugado como método de otimização e Newton como método de busca linear na forma irrestrita, na forma restrita foi utilizado o gradiente conjugado como método de otimização e o método quadrático para cálculo do passo, com ponto inicial $\mathbf{x}_0 = [2.5, 0.5, 2, -1, 0.5]^T$.

$$\begin{aligned} \min \quad & f_{22}(\mathbf{x}) = (x_1 - x_2)^2 + (x_2 + x_3 - 2)^2 + (x_4 - 1)^2 + (x_5 - 1)^2 \\ \text{s.a} \quad & x_1 + 3x_2 - 4 = 0 \\ & x_3 + x_4 - 2x_5 = 0 \\ & x_2 - x_5 = 0 \\ & \mathbf{x} = [2,5 \;\; 0,5 \;\; 2 \;\; -1 \;\; 0,5]^t. \end{aligned}$$

Foram obtidos os seguintes resultados na forma irrestrita:

```
4: mu: 0.1, a(x): 0, mu*a(x): 0
0: { 2.5, 0.5, 2, -1, 0.5 }
1: { 1.27143, 1.42143, 1.69286, 0.228571, 0.807143 }
2: { 0.999991, 1.00001, 0.999998, 1.00001, 1 }
3: { 1, 1, 1, 1, 1 }
x_final = { 1, 1, 1, 1, 1 }
f_final = 1.6356e-20
steps = 4
||grad(x)||: 2.55781e-10
g(x) < 0: true
```

Foram obtidos os seguintes resultados na forma restrita:

```
18: mu: 0.1, a(x): 4.56634e-11, mu*a(x): 4.56634e-12
0: { 2.5, 0.5, 2, -1, 0.5 }
1: { 1.29721, 1.4021, 1.6993, 0.202795, 0.800699 }
2: { 0.858733, 0.950171, 1.09039, 0.962961, 1.04771 }
3: { 0.915601, 0.918015, 1.10214, 0.942831, 1.0172 }
4: { 0.904997, 0.907388, 1.08293, 0.991237, 1.01208 }
5: { 0.906018, 0.910867, 1.08495, 0.993942, 1.01095 }
6: { 0.912597, 0.912315, 1.08442, 0.996142, 1.01076 }
7: { 0.931569, 0.93086, 1.07228, 0.994719, 1.01435 }
8: { 0.979718, 0.973463, 1.02245, 0.994688, 1.00666 }
9: { 0.99885, 0.999298, 1.0011, 0.997439, 0.999736 }
10: { 1.00043, 1.00021, 0.99942, 0.999773, 0.999523 }
11: { 1.00027, 1.00036, 0.999594, 0.999972, 0.999877 }
12: { 1.00019, 1.00015, 0.999677, 1.00006, 1.00009 }
13: { 0.999997, 1.00005, 0.999971, 1.00002, 1.00008 }
14: { 0.999987, 0.999986, 1, 1, 1.00004 }
15: { 0.999986, 0.999996, 1.00001, 1, 1.00001 }
16: { 0.99999, 0.99999, 1.00001, 1, 1 }
17: { 0.99999, 0.99999, 1.00001, 0.999999, 1 }
x_final = { 0.99999, 0.99999, 1.00001, 0.999999, 1 }
f_final = 2.13915e-12
steps = 18
||grad(x)||: 3.70778e-06
g(x) < 0: true
```

Ao final do teste irrestrito, substituímos o ponto ótimo nas restrições de desigualdade para averiguar o nível de violação das restrições. A violação se manteve num nível de magnitude próximo do erro demandado.

```
h(x): { -4.03647e-05 6.75747e-06 -1.11548e-05 }
```

Vamos repetir o teste com erro $\varepsilon = 10^{-10}$ para observar se há uma melhora no ótimo ou uma diminuição da violação.

```
41: mu: 0.1, a(x): 2.74614e-14, mu*a(x): 2.74614e-15
0: { 2.5, 0.5, 2, -1, 0.5 }
1: { 1.29721, 1.4021, 1.6993, 0.202795, 0.800699 }
2: { 0.858733, 0.950171, 1.09039, 0.962961, 1.04771 }
3: { 0.915601, 0.918015, 1.10214, 0.942831, 1.0172 }
4: { 0.904997, 0.907388, 1.08293, 0.991237, 1.01208 }
5: { 0.906018, 0.910867, 1.08495, 0.993942, 1.01095 }
6: { 0.912597, 0.912315, 1.08442, 0.996142, 1.01076 }
7: { 0.931569, 0.93086, 1.07228, 0.994719, 1.01435 }
8: { 0.979718, 0.973463, 1.02245, 0.994688, 1.00666 }
```

```
9: { 0.99885, 0.999298, 1.0011, 0.997439, 0.999736 }
10: { 1.00043, 1.00021, 0.99942, 0.999773, 0.999523 }
11: { 1.00027, 1.00036, 0.999594, 0.999972, 0.999877 }
12: { 1.00019, 1.00015, 0.999677, 1.00006, 1.00009 }
13: { 0.999997, 1.00005, 0.999971, 1.00002, 1.00008 }
14: { 0.999987, 0.999986, 1, 1, 1.00004 }
15: { 0.999986, 0.999996, 1.00001, 1, 1.00001 }
16: { 0.99999, 0.99999, 1.00001, 1, 1 }
17: { 0.99999, 0.99999, 1.00001, 0.999999, 1 }
18: { 0.99999, 0.99999, 1.00001, 0.999999, 1 }
19: { 0.999993, 0.999993, 1.00001, 1, 1 }
20: { 0.999998, 0.999997, 1, 1, 1 }
21: { 1, 1, 1, 1, 1 }
...
40: { 1, 1, 1, 1, 1 }
x_final = { 1, 1, 1, 1, 1 }
f_final = 1.70301e-14
steps = 41
||grad(x)||: 2.69691e-07
g(x) < 0: true

h(x): { -1.90534e-06 1.65715e-07 -5.97662e-07 }
```

Não houve uma melhora relativamente significativa da violação das restrições com a diminuição do erro tolerado.

Ao executarmos com outros métodos tanto de busca como de otimização percebemos que de fato estes parâmetros foram os que alcançaram melhores resultados em termos de velocidade de compilação, mesmo com muitos pontos repetidos na reta final do algoritmo o resultado ainda se manteve a frente da utilização dos métodos de busca de Newton e Armijo e do método de otimização quase-newton.

3.2.3 Terceiro problema teste

Este problema teste foi executado utilizando o gradiente conjugado como método de otimização e Newton como método de busca linear na forma irrestrita, na forma restrita foi utilizado o gradiente conjugado como método de otimização e o método quadrático e o método quadrático para cálculo do passo, com ponto inicial $\mathbf{x}_0 = [2, 2]$

$$\begin{aligned} \min \quad & f_{23}(\mathbf{x}) = (x_1 - 2)^2 + (x_2 - 1)^2 \\ \text{s.a} \quad & 0.25x_1^2 + x_2^2 - 1 \leq 0 \\ & x_1 - 2x_2 + 1 = 0 \\ & \mathbf{x}^0 = [2 \;\; 2]^t. \end{aligned}$$

Foram obtidos os seguintes resultados na forma irrestrita:

```
2: mu: 0.1, a(x): 0, mu*a(x): 0
0: { 2, 2 }
1: { 2, 1 }
x_final = { 2, 1 }
f_final = 0
steps = 2
||grad(x)||: 2.48253e-16
g(x) < 0: true
```

Como na forma restrita foram muitos pontos encontrados, mostraremos apenas as informações dos sub-problemas:

```
10: mu: 0.1, a(x): 1.41703, mu*a(x): 0.141703
18: mu: 1, a(x): 0.313895, mu*a(x): 0.313895
26: mu: 10, a(x): 0.0112351, mu*a(x): 0.112351
36: mu: 100, a(x): 0.000144128, mu*a(x): 0.0144128
46: mu: 1000, a(x): 1.48326e-06, mu*a(x): 0.00148326
56: mu: 10000, a(x): 1.48759e-08, mu*a(x): 0.000148759
66: mu: 100000, a(x): 1.48803e-10, mu*a(x): 1.48803e-05
77: mu: 1e+06, a(x): 1.64892e-12, mu*a(x): 1.64892e-06
87: mu: 1e+07, a(x): 5.47866e-14, mu*a(x): 5.47866e-07
111: mu: 1e+08, a(x): 8.19073e-16, mu*a(x): 8.19073e-08
146: mu: 1e+09, a(x): 8.39651e-18, mu*a(x): 8.39651e-09
191: mu: 1e+10, a(x): 1.04223e-19, mu*a(x): 1.04223e-09
226: mu: 1e+11, a(x): 6.43002e-22, mu*a(x): 6.43002e-11
x_final = { 0.822874, 0.911438 }
f_final = 1.39347
steps = 226
||grad(x)||: 2.36091
g(x) < 0: false

h(x): { -1.9733e-06 }
```

Ao se testar outros métodos, encontraremos uma quantidade de iterações maior ou menor, também é importante lembrar que ao se alterar parâmetros como precisão e erro também será alterada o número de iterações e de pontos encontrados pelo algoritmo. Portanto é importante que haja um intensivo teste de todas as possibilidades de combinação de parâmetros, para que se encontre a melhor maneira de executar cada função, obtendo assim os pontos e valor ótimo na menor quantidade de tempo e iterações possíveis.

3.3 Métodos de barreira e de penalidade

Por fim, os próximos três problemas serão solucionados por ambos os métodos. Os parâmetros padrão mencionados anteriormente serão utilizados, a não ser que seja indicado o contrário.

3.3.1 Primeiro problema teste

$$
\begin{aligned}
\min \quad & f_{31}(\mathbf{x}) = x_1 x_4 (x_1 + x_2 + x_3) + x_3 \\
\text{s.a} \quad & x_1 x_2 x_3 x_4 - 25 \geq 0 \\
& x_1^2 x_2^2 + x_3^2 x_4^2 - 40 \geq 0 \\
& 1 \leq x_i \leq 5, \quad \forall i \in \{1, 2, 3, 4\} \\
& \mathbf{x} = [1 \;\; 5 \;\; 5 \;\; 1]^t
\end{aligned}
$$

Ao executar o teste irrestrito, utilizamos os seguintes parâmetros: método irrestrito: gradientes conjugados; método de busca linear: Armijo;

```
5543: mu: 0.1, a(x): 0, mu*a(x): 0
x_final = { -2.35839e-11, 571.116, 4.13054e+08, 7.73718e-11 }
f_final = 4.13054e+08
steps = 5543
||grad(x)||: 1.01503
g(x) < 0: true
```

A função não aparenta possuir mínimos locais. Prosseguindo paras métodos restritos. Ao executar o teste de barreira, utilizamos os seguintes parâmetros: método irrestrito: gradientes conjugados; método de busca linear: quadrático;

```
37: mu: 10, B(x): -9.36591, mu*B(x): -93.6591,
99: mu: 1, B(x): -2.8134, mu*B(x): -2.8134,
155: mu: 0.1, B(x): 3.23025, mu*B(x): 0.323025,
299: mu: 0.01, B(x): 12.3035, mu*B(x): 0.123035,
392: mu: 0.001, B(x): 25.8425, mu*B(x): 0.0258425,
399: mu: 0.0001, B(x): 33.931, mu*B(x): 0.0033931,
401: mu: 1e-05, B(x): 37.8077, mu*B(x): 0.000378077,
402: mu: 1e-06, B(x): 37.8077, mu*B(x): 3.78077e-05,
403: mu: 1e-07, B(x): 37.8077, mu*B(x): 3.78077e-06,
404: mu: 1e-08, B(x): 37.8077, mu*B(x): 3.78077e-07,
400: { 1.00002, 4.99999, 4.9912, 1.00174 }
401: { 1.00002, 4.99999, 4.9912, 1.00174 }
402: { 1.00002, 4.99999, 4.9912, 1.00174 }
403: { 1.00002, 4.99999, 4.9912, 1.00174 }
x_final = { 1.00002, 4.99999, 4.9912, 1.00174 }
f_final = 16.0018
steps = 404
||grad(x)||: 16.4351
g(x) < 0: true
```

Infelizmente a função se mostrou mau comportada para a nosso método de penalidade. O termo de penalidade $\mu\alpha(\mathbf{x}_k)$ né crescente neste caso, pois mesmo fazendo $\beta \to 1$ (empiricamente), o parâmetro μ cresce mais rápido do que $\alpha(\mathbf{x}_\mu)$ diminui para cada μ.

Qualquer viés que podíamos ter favorecendo o método da penalidade com relação ao método de barreira foi destruído com este experimento.

3.3.2 Segundo problema teste

$$
\begin{aligned}
\min \quad & f_{32}(\mathbf{x}) = x_1 - x_2 - x_3 - x_1x_3 + x_1x_4 + x_2x_3 - x_2x_4 \\
\text{s.a} \quad & 8 - x_1 - 2x_2 \geq 0 \\
& 12 - 4x_1 - x_2 \geq 0 \\
& 12 - 3x_1 - 4x_2 \geq 0 \\
& 8 - 2x_3 - x_4 \geq 0 \\
& 8 - x_3 - 2x_4 \geq 0 \\
& 5 - x_3 - x_4 \geq 0 \\
& x_i \geq 0, \quad \forall i \in \{1, 2, 3, 4\} \\
& \mathbf{x} = [0 \;\; 0 \;\; 0 \;\; 0]^t.
\end{aligned}
$$

Ao executar o teste irrestrito, utilizamos os seguintes parâmetros: método irrestrito: quase-Newton; método de busca linear: Newton;

```
0: { 1, 1, 1, 1 }
1: { 0.25, 1.75, 1.75, 1 }
x_final = { 0.25, 1.75, 1.75, 1 }
f_final = -2.125
steps = 2
||grad(x)||: 1.62019
g(x) < 0: true
```

Verificamos se este ponto está no conjunto viável executando o seguinte código,

```
cout << "g(x) <= 0: " << check_inequality_restrictions(
    inequality_restrictions8, data81.x_final) << endl;
cout << "g(x): {";
for (auto restriction : inequality_restrictions8)
    cout << restriction(data81.x_final) << " ";
cout << "}";
```

cujo resultado é

```
g(x) <= 0: 1
g(x): { -4.25 -9.25 -4.25 -3.5 -4.25 -2.25 -0.25 -1.75 -1.75 -1 }
```

confirmando a viabilidade do ótimo globalmente encontrado.

Ao executar o teste de barreira, utilizamos os seguintes parâmetros: método irrestrito: gradientes conjugados; método de busca linear: Armijo;

```
27: mu: 1, B(x): -3.9755, mu*B(x): -3.9755,
57: mu: 0.1, B(x): 6.40243, mu*B(x): 0.640243,
```

```
76: mu: 0.01, B(x): 15.7003, mu*B(x): 0.157003,
93: mu: 0.001, B(x): 24.9167, mu*B(x): 0.0249167,
104: mu: 0.0001, B(x): 34.1743, mu*B(x): 0.00341743,
106: mu: 1e-05, B(x): 38.0638, mu*B(x): 0.000380638,
107: mu: 1e-06, B(x): 38.0638, mu*B(x): 3.80638e-05,
108: mu: 1e-07, B(x): 38.0638, mu*B(x): 3.80638e-06,
109: mu: 1e-08, B(x): 38.0638, mu*B(x): 3.80638e-07,

x_final = { 3.55033e-07, 2.99998, 2.35641e-05, 3.99996 }
f_final = -14.9997
steps = 109
||grad(x)||: 7.93716
g(x) < 0: true
```

Porém, repetindo o teste desta vez utilizando uma barreira inversa (2.3), melhoramos nosso resultado:

```
15: mu: 1, B(x): -9.11524e+06, mu*B(x): -9.11524e+06,
16: mu: 0.1, B(x): -9.11524e+06, mu*B(x): -911524,
17: mu: 0.01, B(x): -9.11524e+06, mu*B(x): -91152.4,
18: mu: 0.001, B(x): -9.11524e+06, mu*B(x): -9115.24,
19: mu: 0.0001, B(x): -9.11524e+06, mu*B(x): -911.524,
20: mu: 1e-05, B(x): -9.11524e+06, mu*B(x): -91.1524,
21: mu: 1e-06, B(x): -9.11524e+06, mu*B(x): -9.11524,
22: mu: 1e-07, B(x): -9.11524e+06, mu*B(x): -0.911524,
23: mu: 1e-08, B(x): -9.11524e+06, mu*B(x): -0.0911524,
24: mu: 1e-09, B(x): -9.11524e+06, mu*B(x): -0.00911524,
25: mu: 1e-10, B(x): -9.11524e+06, mu*B(x): -0.000911524,
26: mu: 1e-11, B(x): -9.11524e+06, mu*B(x): -9.11524e-05,
27: mu: 1e-12, B(x): -9.11524e+06, mu*B(x): -9.11524e-06,
28: mu: 1e-13, B(x): -9.11524e+06, mu*B(x): -9.11524e-07,
x_final = { 1.11834e-07, 0.129646, 0.000227017, 3.99988 }
f_final = -0.648413
steps = 28
||grad(x)||: 7.12513
g(x) < 0: true
```

Ressaltamos então a importância de ajustar os parâmetros de otimização para cada problema.

Ao executar o teste de penalidade, utilizamos os seguintes parâmetros: método irrestrito: gradientes conjugados; método de busca linear: Armijo;

Entretanto, mais uma vez o método de penalidade se mostrou instável na solução do problema. Para todos os valores válidos dos parâmetros μ, β e p testados, ocorre um ponto de inflexão na função $\mu\alpha(\mathbf{x}_\mu)$ em que μ passa a crescer mais rápido que $\alpha(\mathbf{x}_\mu)$. Exemplificamos o fenômeno utilizando um parâmetro alto $p = 10$.

```
575: mu: 1e+38, a(x): 1.53333e-42, mu*a(x): 0.000153333
582: mu: 1e+39, a(x): 8.45907e-44, mu*a(x): 8.45907e-05
586: mu: 1e+40, a(x): 1.18975e-44, mu*a(x): 0.000118975
590: mu: 1e+41, a(x): 1.92537e-45, mu*a(x): 0.000192537
```

Observe a magnitude explosiva do parâmetro μ, evidenciando as dificuldades numéricas envolvidas no método.

Para obter um resultado, somos então obrigados a aumentar o erro tolerado. Ao executar o teste abaixo, utilizamos os seguintes parâmetros: método irrestrito: gradientes conjugados; método de busca linear: Armijo; $p = 20$; $\mu = 0.1$; $\beta = 10$; **erro:** $\varepsilon = 10 - 5$**.**

```
1356: mu: 1e+70, a(x): 1.1649e-74, mu*a(x): 0.00011649
1357: mu: 1e+71, a(x): 1.1649e-74, mu*a(x): 0.0011649
1358: mu: 1e+72, a(x): 1.1649e-74, mu*a(x): 0.011649
1359: mu: 1e+73, a(x): 1.1649e-74, mu*a(x): 0.11649
1360: mu: 1e+74, a(x): 1.1649e-74, mu*a(x): 1.1649
1361: mu: 1e+75, a(x): 1.1649e-74, mu*a(x): 11.649
1362: mu: 1e+76, a(x): 1.1649e-74, mu*a(x): 116.49
1363: mu: 1e+77, a(x): 1.1649e-74, mu*a(x): 1164.9
1364: mu: 1e+78, a(x): 1.1649e-74, mu*a(x): 11649
1365: mu: 1e+79, a(x): 1.1649e-74, mu*a(x): 116490
1366: mu: 1e+80, a(x): 1.1649e-74, mu*a(x): 1.1649e+06
1382: mu: 1e+81, a(x): 3.81337e-85, mu*a(x): 0.000381337
1384: mu: 1e+82, a(x): 1.73534e-86, mu*a(x): 0.000173534
1387: mu: 1e+83, a(x): 2.74699e-91, mu*a(x): 2.74699e-08
x_final = { -2.90651e-05, 3.00003, 8.73975e-05, 3.99997 }
f_final = -15
steps = 1387
||grad(x)||: 7.93714
g(x) < 0: false
```

Observe que a estabilidade numérica do método foi bastante precária.

3.3.3 Terceiro problema teste

$$
\begin{aligned}
\min \quad & f_{33}(\mathbf{x}) = (x_1 - 10)^2 + 5(x_2 - 12)^2 + x_3^4 + 3(x_4 - 11)^2 \\
& +10x_5^6 + 7x_6^2 + x_7^4 - 4x_6x_7 - 10x_6 - 8x_7 \\
\text{s.a} \quad & 127 - x_1^2 - 3x_2^4 - x_3 - 4x_4^2 - 5x_5 \geq 0 \\
& 282 - 7x_1 - 3x_2 - 10x_3^2 - x_4 + x_5 \geq 0 \\
& 196 - 23x_1^2 - x_2^2 - 6x_6^2 + 8x_7 \geq 0 \\
& -4x_1^2 - x_2^2 + 3x_1x_2 - 2x_3^2 - 5x_6 + 11x_7 \geq 0 \\
& \mathbf{x} = [1 \quad 2 \quad 0 \quad 4 \quad 0 \quad 1 \quad 1]^t
\end{aligned}
$$

Ao executar o teste irrestrito, utilizamos os seguintes parâmetros: método irrestrito: gradientes conjugados; método de busca linear: Armijo;

```
36: mu: 10, B(x): -0, mu*B(x): -0,
0: { 1, 2, 0, 4, 0, 1, 1 }
1: { 1.13282, 2.73787, 0, 2.42096, 0, 1, 1.05903 }
2: { 2.57868, 10.2634, 0, 2.88864, 0, 1.02028, 1.64727 }
3: { 2.83179, 10.7177, 0, 2.21323, 0, 1.05405, 1.57966 }
...
33: { 10, 12, 0, 2.34984, 0, 1.13233, 1.46315 }
34: { 10, 12, 0, 2.34984, 0, 1.13233, 1.46315 }
35: { 10, 12, 0, 2.34984, 0, 1.13233, 1.46315 }
x_final = { 10, 12, 0, 2.34984, 0, 1.13233, 1.46315 }
f_final = 238.868
steps = 36
||grad(x)||: 6.30318e-06
g(x) < 0: true
```

Verificamos também a viabilidade do ótimo global:

```
g(x) <= 0: 0
g(x): { 26 40 66 -5.65016 -3.30032 -2.65016 -10 -12 -0 -2.34984 }
```

Ao executar o teste de barreira os seguintes parâmetros que renderam os melhores resultados foram: método irrestrito: gradientes conjugados; método de busca linear: Armijo; tipo de barreira: logarítmica;

```
2: mu: 1, B(x): -13.4928, mu*B(x): -13.4928,
12: mu: 0.1, B(x): -6.48011, mu*B(x): -0.648011,
16: mu: 0.01, B(x): -6.43821, mu*B(x): -0.0643821,
20: mu: 0.001, B(x): -5.40343, mu*B(x): -0.00540343,
22: mu: 0.0001, B(x): -2.86116, mu*B(x): -0.000286116,
23: mu: 1e-05, B(x): -2.86116, mu*B(x): -2.86116e-05,
24: mu: 1e-06, B(x): -2.86116, mu*B(x): -2.86116e-06,
25: mu: 1e-07, B(x): -2.86116, mu*B(x): -2.86116e-07,
x_final = { 1.24426, 2.40973, -0.00669594, 2.47333, -0.0334695,
            0.999254, 1.1169 }
f_final = 777.218
steps = 25
||grad(x)||: 98.1483
g(x) < 0: true
```

Ao executar o teste de penalidade, mais uma vez tivemos que aumentar o nosso erro. Utilizamos os seguintes parâmetros: método irrestrito: gradientes conjugados; método de busca linear: Armijo; erro: 10^{-5};

```
10: mu: 0.1, a(x): 1.0029, mu*a(x): 0.10029
21: mu: 1, a(x): 0.0255861, mu*a(x): 0.0255861
23: mu: 10, a(x): 0.0101198, mu*a(x): 0.101198
30: mu: 100, a(x): 2.15848e-08, mu*a(x): 2.15848e-06
```

```
0: { 1, 2, 0, 4, 0, 1, 1 }
1: { 1.10625, 2.5903, 0, 2.73677, 0, 1, 1.04722 }
2: { 1.23094, 2.65625, -0.0031836, 1.17356, -0.015918,
     1.00008, 1.10557 }
...
27: { 2.56841, 2.42882, -0.0998241, 2.16706, -0.534619,
      0.923692, 1.65406 }
28: { 2.56841, 2.42877, -0.0998246, 2.16706, -0.53462,
      0.92369, 1.65407 }
29: { 2.56771, 2.42877, -0.0998215, 2.16726, -0.534644,
      0.923365, 1.65492 }
x_final = { 2.56771, 2.42877, -0.0998215, 2.16726, -0.534644,
      0.923365, 1.65492 }
f_final = 754.512
steps = 30
||grad(x)||: 97.9514
g(x) < 0: false

h(x): { 0.000146918 -253.938 -46.583 -0.00504091 }
```

Curiosamente, observamos uma pequena violação da primeira restrição.

4 Manual da Implementação

Este sendo o terceiro trabalho da disciplina, ele foi construído totalmente sobre o trabalho feito anteriormente, no nosso caso em Python (através de *notebooks* da Google Colab). Por isso, em relação ao que já tinha sido implementado anteriormente, o código fonte atual tenta traduzir para o C++ a exata implementação das funções definidas em Python. No entanto, ainda é preciso se adaptar à diferenças entre as duas linguagens, especialmente no que diz respeito ao acesso e apresentação dos dados, em que Python é muito mais amigável com o usuário. Por exemplo, não existe por padrão uma funcionalidade de *printar* um array ou um vector diretamente no console.

4.1 O pacote linear_algebra .hh

O projeto só faria sentido se fosse possível aproximar a interface construída em Python dentro do C++. Assim, foi elaborado o pacote linear_algebra.hh. Ele implementa funcionalidades de álgebra linear e de qualidade de vida para inerfaces numéricas. Como esse é um tipo de projeto que pode ficar para a posteridade, ele foi todo implementado em inglês. Segue uma lista das principais funcionalidades implementadas nesse pacote:

- Capacidade de *printar* vetores e matrizes
- Operações unárias de álgebra linear (transposição, norma, inversão de sinal etc)
- Operações binárias de álgebra linear com linguagem simples. A intenção é reproduzir a naturalidade de operações matriciais de pacotes como o Numpy ou o MATLAB. Isto é, o produto interno entre dois vetores v e u deve ser escrito simplesmente utilizando o operador $*$ utilizado para multiplicação: **double** w = v $*$ u;
- Utilidades. Exemplos: geração automática de vetores do tipo $\{a, a+1 \ldots, b-1\}$ para argumentos a e b; geração automática de objetos representando a matriz identidade I_n para o argumento $n \in \mathbb{Z}^+$; rotina de inversão de matrizes etc

Observação Todas as definições inclusas nesse pacote são paramétricas, de modo que o cabeçalho de definição do template (**template** <**typename** T>) será omitido neste documento.

Devido a possibilidade de o usuário encontrar utilidade em utilizar as definições desse pacote e pela necessidade de documentação do mesmo, uma breve descrição será feita das funções e objetos mais importantes definidos.

Observação Todos os métodos e funções retornam uma nova cópia do objeto, deixando os originais intactos.

Matrix<T> Define um objeto representando uma matriz que se comporta quase exatamente como vector<vector<T>>, cujos elementos podem ser acessados pelo operador colchete (matriz[i][j] ou matriz[i]) ou parêntese (matriz(i, j)).

Construtores:

- Matrix()
- Matrix(std::vector<T> user_v, **bool** column = **true**)
- Matrix(std::vector<std::vector<T>> user_m)
- Matrix(Matrix<T>* user_m)
- Matrix(size_t m, size_t n)

Métodos notáveis da classe:

size_t n_rows() A quantidade de linhas da matriz

Matrix<T> concatenate(Matrix<T>) other Concatena duas matrizes

Matrix<T> slice(size_t, size_t, size_t, size_t) Retorna uma submatriz

Matrix<T> inverse() Retorna a matriz inversa

Matrix<T> t(Matrix<T> M) Retorna a matriz transposta

double norm() Retorna a norma caso a matriz seja linha ou coluna

static double norm(std::vector<T> v) Sobrecarga do método anterior

Matrix<T> I(size_t m) Retorna uma matriz identidade de dimensão m

std::vector<T> range(unsigned int n) Retorna um vetor $\{1, \ldots, n\}$

Matrix<T> t(std::vector<T> v) Retorna uma matriz coluna a partir do vetor **v**

Sobrecarga de operadores Operadores sobrecarregados: $+ \; - \; * \; += \; -= \; *= \; <<$

As outras definições importantes dizem respeito à sobrecarga dos operadores aritméticos $+ \; - \; *$. No caso dos operadores adição, subtração e sinal negativo (existe uma diferença entre os dois últimos em programação), todas as operações se comportam como o esperado para álgebra linear. No caso do operador $*$ a filosofia adotada foi de que onde possível a operação retornaria um tipo mais básico. Por exemplo, se v e u são do tipo vector<T> e de mesmo comprimento, v * u retorna um escalar do tipo T (int, float, double etc). A exata precedência de operações de conversão de tipos não será abordada aqui, mas o usuário deve manter essa estrutura em mente caso utilize essas operações. Por fim, o operador << emite corretamente objetos do tipo vector<T> e Matrix<T> para a *stream* designada.

4.2 O pacote restricted_optimization .hh

Por ser um trabalho derivado de anteriores, naturalmente toda a funcionalidade presente nos programas anteriores está presente aqui. Existem funções para métodos de busca linear do passo, funções que retornam outras funções que aproximam numericamente o gradiente e a hessiana de alguma função matemática e os métodos de otimização irrestrita implementados anteriormente (à exceção do método do gradiente), todas com lista de parâmetros quase idênticas à implementação anterior. Funcionalidades que ficaram de fora foram a habilidade de processar uma *string* diretamente através de matemática simbólica e a habilidade de *plotar* os gráficos de contorno e trajetória. Ambas as funcionalidades são mais bem vindas em outras plataformas.

No espírito da brevidade, vamos descrever apenas as funções mais relevantes para o tema deste trabalho, as funções que solucionam um problema restrito. Porém, antes disso, faz-se necessário descrever as definições de tipo feitas pelo pacote.

- **typedef** vector<**double**> vecd; (por praticidade)
- **typedef** function<**double**(vecd)> real_function;

  Uma função real de uma variável vetorial. Tem um argumento do tipo vecd e retorna um valor do tipo **double**. Seu uso será para guardar as diversas funções $f(\mathbf{x})$, $g_i(\mathbf{x})$, $h_i(\mathbf{x})$ mencionadas anteriormente.
- **typedef** function<vecd(vecd)> vector_function;

  Uma função vetorial de uma variável vetorial. Tem um argumento do tipo vecd e retorna um valor do tipo vecd. Seu uso será para guardar a função gradiente $F(\mathbf{x}) = \nabla f(\mathbf{x})$ de uma função $f(\mathbf{x})$.
- **typedef** function<Matrix<**double**>(vecd)> matrix_function; Uma função matricial de uma variável vetorial. Tem um argumento do tipo vecd e retorna um valor do tipo Matrix<**double**>. Seu uso será para guardar a função matriz hessiana $H(\mathbf{x}) = \nabla^2 f(\mathbf{x})$ de uma função $f(\mathbf{x})$.

4.2.1 Método de barreira: minimize_barrier_method

Observação As funções para o método de barreira e para o método de penalidade (minimize_penalty_method) são extremamente similares.

Cabeçalho e descrição dos parâmetros

```
void minimize_barrier_method (real_function func, int n,
vector<real_function> restrictions, vecd x_initial,
vecd& x_final, double& f_final, Matrix<double>& trajectory,
string minimization_method = "conjugate-gradient",
string linear_search_method = "newton",
string barrier_type = "log", double mu = 10, double beta = 0.1,
```

double error = 1e−6, **double** precision = 1e−6, **double** rho = 1, **double** eta = 0.5, **double** sigma = 0)

Parâmetro	Tipo	Descrição
func	real_function	Uma função que retorna o valor da função objetivo $f(\mathbf{x})$ no ponto $\mathbf{x}$.
int	n	A dimensionalidade do problema.
restrictions	vector<real_function>	Um vector cujos elementos são funções representando cada restrição de desigualdade.
x_initial	vecd	Um vector<double> contendo $\mathbf{x}_1 = (x_1^1, \ldots, x_n^1)$, o ponto inicial.
x_final	vecd&	Uma referência ao vecd que guardará o ótimo $\overline{\mathbf{x}}$.
f_final	double&	Uma referência à variável double que guardará o valor ótimo $f(\overline{\mathbf{x}})$.
trajectory	Matrix<double>&	Uma referência a um objeto tipo matriz ($k \times n$, onde k é o número total de passos gerados) cuja utilidade será armazenar a sequência de pontos $\{\mathbf{x}_k\}$ gerada pelos métodos de minimização.

Parâmetros opcionais		
minimization_method	string	O método de otimização irrestrita e ser empregado nos passos. Opções: "newton-method", "conjugate-gradient", "quasi-newton".
linear_search_method	string	O método de busca linear do passo a ser empregado. Opções: "newton", "armijo", "golden-section". No caso de escolha do método irrestrito "conjugate-gradient", também está disponível a opção "quadratic".
barrier_type	string	O tipo de função barreira a ser empregada. Opções: "log"e "inverse".
mu	double	Valor inicial do parâmetro de "penalidade" μ.
beta	double	Valor de escala do parâmetro μ, β.
error	double	Tolerância para parada dos métodos de otimização.
precision	double	Intervalo empregado nos diversos métodos de diferenciação.
rho	double	Parâmetro de escala do intervalo do método da seção áurea, ρ.
eta	double	Parâmetro de escala do passo no método de busca de Armijo, η.

sigma	double	Tem diferentes significados dependendo do método de otimização irrestrita escolhido. Newton: nível de aproximação do método do gradiente (não recomendável $\sigma > 0$ para otimização com restrições). Quase-Newton: multiplicador da matriz estimativa inicial (I_n) da hessiana de $f(x)$.

Valor de retorno e efeitos

A função não tem nenhum valor de retorno. Ao ser chamada, ela invoca um dos métodos de otimização restrita (padrão gradientes conjugados) com os parâmetros escolhidos para resolver uma sequência de problemas irrestritos da forma

$$\begin{aligned} \text{minimizar} \quad & f(\mathbf{x}) + \mu B(\mathbf{x}) \\ \text{s.a.} \quad & \mathbf{g}(\mathbf{x}) < \mathbf{0} \\ & \mathbf{x} \in X, \end{aligned}$$

onde $B(\mathbf{x})$ é como em 2.3.

Ela armazena os dados gerados pelo método nas variáveis passadas por referência. O último ótimo $\mathbf{x}_{\mu_k}$ encontrado é armazenado em x_final, o valor da função objetivo original no ponto $f(\mathbf{x}_{\mu_k})$ é armazenado em f_final e a sequência de todos os pontos visitados (não apenas os ótimos parciais) $\{x_k\}$ é armazenada em trajectory.

A função *printa* no console a seguinte informação:

$$k \text{ (passo total atual)}, \ \mu_k, \ B(\mathbf{x}_k), \ \mu B(\mathbf{x}_k), \quad \text{ou } \mathbf{false},$$

a cada vez que um ótimo de um problema irrestrito é encontrado. Em seguida o valor de μ é atualizado e o método avança para o próximo problema irrestrito da sequência. Esta informação pode ser útil para verificar a saúde do método e a evolução da sequência gerada. Por exemplo:

```
23: mu: 10, B(x): -4.93167, mu*B(x): -49.3167,
40: mu: 1, B(x): 3.31516, mu*B(x): 3.31516,
43: mu: 0.1, B(x): 3.42778, mu*B(x): 0.342778,
49: mu: 0.01, B(x): 3.77674, mu*B(x): 0.0377674,
50: mu: 0.001, B(x): 3.77674, mu*B(x): 0.00377674,
51: mu: 0.0001, B(x): 3.77674, mu*B(x): 0.000377674,
52: mu: 1e-05, B(x): 3.77674, mu*B(x): 3.77674e-05,
53: mu: 1e-06, B(x): 3.77674, mu*B(x): 3.77674e-06,
54: mu: 1e-07, B(x): 3.77674, mu*B(x): 3.77674e-07,
```

Esta saída indica, pela sequência de números consecutivos $49, 50, 51, 52, 53, 54$, que a fronteira foi atingida antes do critério de terminação escolhido (erro = 1e−6, $\varepsilon = 1 \times 10^{-6}$) ser satisfeito, de modo que o algoritmo tentou atualizar μ e resolver um novo problema irrestrito, porém sem conseguir melhorar a solução obtida, até que o critério de terminação foi satisfeito.

Modo de uso

O usuário deve preparar as variáveis a serem passados como argumentos obrigatórios para a função. O usuário deve declarar uma função que receba um vetor vecd e retorne **double** representando a função objetivo. O usuário também deve declarar uma lista de funções do mesmo tipo anterior e com as mesmas convenções para representar cada uma das restrições. Essa lista deve ser um contêiner do tipo vector contendo referências para as funções que representam as restrições, ou objetos com qualidade de função, por exemplo, o tipo conveniente real_function ou funções lambda. Por fim, o usuário deve declarar as variáveis que receberão os dados gerados pelo método.

Note que a convenção utilizada nesse pacote é de que os pontos são tratados como vetores e implementados como vecd, de modo que as suas componentes, ou as variáveis, são acessadas como elementos desse vetor (ex: x[0], ..., x[n]).

Observação Todas as restrições devem ser de negatividade, ou seja, do tipo $g_i(\mathbf{x}) \leq 0$. Caso o problema envolva restrições do tipo contrário, é necessário multiplicar essas restrições por -1.

O modo mais básico de inicializar as variáveis necessárias é através de declarações simples. Após declarar as restrições, o usuário pode colecionar os ponteiros para essas funções num vetor do tipo vector<real_function>. A seguir basta chamar a função com os argumentos declarados e acessar os dados gerados posteriormente. Caso queira, nessa etapa o usuário pode ajustar os outros parâmetros de otimização, como μ e β. Esse procedimento está exemplificado na figura 4.1. No entanto, esse não é o modo mais recomendado de uso, pois não favorece organização. Como o C++ não permite declarações explícitas de funções aninhadas, esse tipo de declaração de função precisa ser feito fora da função main. É recomendado que se utilize o tipo prático real_function em conjunto com funções lambda (ou com outro tipo de instanciação de função). Também é possível utilizar a palavra chave **auto** nesse contexto. Um exemplo de uso recomendado é exibido na figura 4.2.

4.2.2 Método de penalidade: minimize_penalty_method

Resumo

O uso desta função se diferencia da minimize_barrier_method apenas por dois parâmetros. O usuário deve seguir os mesmos passos e esperar os mesmos tipos de resultado, porém deve declarar uma lista extra de funções que retornam **double** e recebem vecd, representando as restrições de igualdade, ausentes no método anterior. Ao invés de especificar o tipo de barreira, o usuário pode especificar a potência p a ser empregada nas funções de penalidade $\alpha(\mathbf{x})$ de acordo com as equações 1.1. O padrão é $p = 2$. Além disso, os papéis dos parâmetros mu (μ) e beta (β) estão invertidos (escolhe-se $\mu > 0$ e $\beta > 1$.

$$\begin{aligned} \min \quad & f(\mathbf{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \\ \text{s.a} \quad & x_1 - x_2^2 \leq 0 \\ & x_1^2 - x_2 \geq 0 \\ & \mathbf{x}_0 = [-2 \;\; 1]^t. \end{aligned}$$

```
#include "restricted_optimization.hh"
using std::pow;

vecd x_initial = { -2, 1 };
double function (vecd x)
{
    return 100*pow(x[1] - pow(x[0], 2), 2) + pow(1 - x[0], 2);
}
double restriction1 (vecd x)
{
    return x[0] - x[1]*x[1];
}
double restriction2 (vecd x)
{
    return -(x[0]*x[0]) + x[1];
}
vector<real_function> restrictions = { restriction1,
                                       restriction2 };

int main()
{
    vecd x_final;
    double f_final;
    Matrix<double> trajectory;
    minimize_barrier_method(function, 2, restrictions, x_initial,
                            x_final, f_final, trajectory);
}
```

Figura 4.1: Método de barreira: exemplo de uso de forma básica.

$$
\begin{aligned}
\min \quad & f(\mathbf{x}) = x_1^2 + 0,5x_2^2 + x_3^2 + 0,5x_4^2 - x_1x_3 + x_3x_4 - x_1 - 3x_2 + x_3 - x_4 \\
\text{s.a} \quad & 5 - x_1 - 2x_2 - x_3 - x_4 \geq 0 \\
& 4 - 3x_1 - x_2 - 2x_3 + x_4 \geq 0 \\
& x_2 + 4x_3 - 1,5 \geq 0 \\
& x_i \geq 0, \quad \forall i \in \{1, 2, 3, 4\} \\
& \mathbf{x}_0 = [0,5 \;\; 0,5 \;\; 0,5 \;\; 0,5]^t.
\end{aligned}
$$

```
#include "restricted-optimization.hh"

int main()
{
    auto function = [](vecd x) {
        return x[0] * x[0] + 0.5 * x[1] * x[1] + x[2] * x[2]
        + 0.5 * x[3] * x[3] - x[0] * x[2] + x[2] * x[3] - x[0]
        - 3 * x[1] + x[2] - x[3];
    };
    vector<real_function> restrictions =
    {
        [](vecd x) { return -5 + x[0] + 2*x[1] + x[2] + x[3]; },
        [](vecd x) { return -4 + 3*x[0] + x[1] + 2*x[2] - x[3]; },
        [](vecd x) { return -x[1] - 4*x[3] + 1.5; },
        [](vecd x) { return -x[0];},
        [](vecd x) { return -x[1];},
        [](vecd x) { return -x[2];},
        [](vecd x) { return -x[3];}
    };
    vecd x_initial = { 0.5, 0.5, 0.5, 0.5 };
    vecd x_final;
    double f_final;
    Matrix<double> trajectory;
    minimize_barrier_method(function, x_initial.size(),
        restrictions, x_initial, x_final, f_final, trajectory);
}
```

Figura 4.2: Método de barreira: exemplo de uso de forma recomendada.

Cabeçalho e descrição dos parâmetros

```
void minimize_penalty_method (real_function func, int n,
    vector<real_function> inequality_restrictions,
    vector<real_function> equality_restrictions,
    vecd x_initial, vecd& x_final, double& f_final,
    Matrix<double>& trajectory,
    string minimization_method = "conjugate-gradient",
    string linear_search_method = "newton", double power = 2,
    double mu = 0.1, double beta = 10, double error = 1e-6,
    double precision = 1e-6, double rho = 1, double eta = 0.5,
    double sigma = 0)
```

Parâmetro	Tipo	Descrição
func	real_function	Uma função que retorna o valor da função objetivo $f(\mathbf{x})$ no ponto $\mathbf{x}$.
int	n	A dimensionalidade do problema.
inequality_restrictions	vector<real_function>	Um vector cujos elementos são funções representando cada restrição de desigualdade.
equality_restrictions	vector<real_function>	Um vector cujos elementos são funções representando cada restrição de igualdade.
x_initial	vecd	Um vector<double> contendo $\mathbf{x}_1 = \left(x_1^1, \ldots, x_n^1\right)$, o ponto inicial.
x_final	vecd&	Uma referência ao vecd que guardará o ótimo $\overline{\mathbf{x}}$.
f_final	double&	Uma referência à variável double que guardará o valor ótimo $f(\overline{\mathbf{x}})$.
trajectory	Matrix<double>&	Uma referência a um objeto tipo matriz ($k \times n$, onde k é o número total de passos gerados) cuja utilidade será armazenar a sequência de pontos $\{\mathbf{x}_k\}$ gerada pelos métodos de minimização.

Parâmetros opcionais		
minimization_method	string	O método de otimização irrestrita e ser empregado nos passos. Opções: "newton-method", "conjugate-gradient", "quasi-newton".

linear_search_method	string	O método de busca linear do passo a ser empregado. Opções: "newton", "armijo", "golden-section". No caso de escolha do método irrestrito "conjugate-gradient", também está disponível a opção "quadratic".
power	double	A potência p a ser empregada nas funções de penalidade $\alpha(\mathbf{x})$.
mu	double	Valor inicial do parâmetro de "penalidade" μ.
beta	double	Valor de escala do parâmetro μ, β.
error	double	Tolerância para parada dos métodos de otimização.
precision	double	Intervalo empregado nos diversos métodos de diferenciação.
rho	double	Parâmetro de escala do intervalo do método da seção áurea, ρ.
eta	double	Parâmetro de escala do passo no método de busca de Armijo, η.
sigma	double	Tem diferentes significados dependendo do método de otimização irrestrita escolhido. Newton: nível de aproximação do método do gradiente (não recomendável $\sigma > 0$ para otimização com restrições). Quase-Newton: multiplicador da matriz estimativa inicial (I_n) da hessiana de $f(x)$.

Valor de retorno e efeitos

A função não tem nenhum valor de retorno. Ao ser chamada, ela invoca um dos métodos de otimização restrita (padrão gradientes conjugados) com os parâmetros escolhidos para resolver uma sequência de problemas irrestritos da forma

$$\begin{aligned} \text{minimizar} \quad & f(\mathbf{x}) + \mu\alpha(\mathbf{x}) \\ \text{s.a.} \quad & \mathbf{x} \in X, \end{aligned}$$

onde $\alpha(\mathbf{x})$ é como em 1.1.

Ela armazena os dados gerados pelo método nas variáveis passadas por referência. O último ótimo $\mathbf{x}_{\mu_k}$ encontrado é armazenado em x_final, o valor da função objetivo original no ponto $f(\mathbf{x}_{\mu_k})$ é armazenado em f_final e a sequência de todos os pontos visitados (não apenas os ótimos parciais) $\{x_k\}$ é armazenada em trajectory.

A função *printa* no console a seguinte informação:

$$k \text{ (passo total atual)}, \ \mu_k, \ \alpha(\mathbf{x}_k), \ \mu\alpha(\mathbf{x}_k),$$

a cada vez que um ótimo de um problema irrestrito é encontrado. Em seguida o valor de μ é atualizado e o método avança para o próximo problema irrestrito da sequência.

Modo de uso

Valem as mesmas convenções utilizadas para a função anterior.

O usuário deve preparar as variáveis a serem passados como argumentos obrigatórios para a função. O usuário deve declarar uma função que receba um vetor vecd e retorne **double** representando a função objetivo. O usuário também deve declarar uma lista de funções do mesmo tipo anterior e com as mesmas convenções para representar cada uma das restrições. Essa lista deve ser um contêiner do tipo vector contendo referências para as funções que representam as restrições, ou objetos com qualidade de função, por exemplo, o tipo conveniente real_function ou funções lambda. Por fim, o usuário deve declarar as variáveis que receberão os dados gerados pelo método. Um exemplo de uso pode ser visto na figura 4.3.

$$
\begin{aligned}
\min \quad & f_{23}(\mathbf{x}) = (x_1 - 2)^2 + (x_2 - 1)^2 \\
\text{s.a} \quad & 0.25x_1^2 + x_2^2 - 1 \leq 0 \\
& x_1 - 2x_2 + 1 = 0 \\
& \mathbf{x}^0 = [2 \;\; 2]^t.
\end{aligned}
$$

```
#include "restricted-optimization.hh"
using std::pow;

int main()
{
    auto function = [](vecd x) { return pow(x[0] - 2, 2) +
                        pow(x[1] - 1, 2); };
    vector<real_function> inequality_restrictions = {
        [](vecd x) { return 0.25*x[0]*x[0] + x[1]*x[1] - 1; }
    };
    vector<real_function> equality_restrictions = {
        [](vecd x) { return x[0] - 2*x[1] + 1; }
    };
    vecd x_initial = { 2, 2 };

    vecd x_final;
    double f_final;
    Matrix<double> trajectory;

    minimize_penalty_method(function, n,
        inequality_restrictions, equality_restrictions,
        x_initial, x_final, f_final, trajectory);
}
```

Figura 4.3: Método de penalidade: exemplo de uso recomendado.

Referências Bibliográficas

[1] BAZARAA, M.; BAZARAA, M.; SHETTY, C.; SHERALI, H.; COLLECTION, K. M. R. *Nonlinear programming: Theory and algorithms*. Wiley, 1979.

[2] RIBEIRO, A.; KARAS, E. *Um curso de otimização*. Curitiba: UFPR, 2011.